SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluating learning algorithms: An efficient way/Efficient ways to find Regular Inductive Statements

Van Tu Nguyen

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Evaluating learning algorithms: An efficient way/Efficient ways to find Regular Inductive Statements

Titel der Abschlussarbeit

Author:	Van Tu Nguyen
Supervisor:	Prof. Dr. Dr. h. c. Javier Esparza
Advisor:	Christoph Welzel-Mohr
Submission Date:	Submission date

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, Submission date Van Tu Nguyen

Abstract

Regular Model Checking, as proposed in the works of [Bou+00], is a widely used paradigm for verifying parameterized and infinite-state systems that occur naturally when a program uses queries, stacks or intergers, etc.

This thesis aims to investigate automata learning, which includes techniques that extract finite automata from external data sources in the domains of verification and synthesis. We are considering an application scenario that is particularly well-suited for Regular Model Checking, where the *regular transition system* models the parameterized systems.

Firtsly, we present an approach for regular model checking that uses inductive statements to over-approximate all reachable states. A statement is inductive if it only relates a state satisfying ϕ with states that also satisfy ϕ. We demonstrate how the statements are encoded and their *interpretations* defined, which helps in understanding the encoded statements.

We will discuss the primary mechanism of learning automata, which involves the use of membership queries and equivalent queries. During the learning process, the Teacher and the Learner interact with each other. The Teacher has knowledge of the target language, while the Learner has the opportunity to ask two types of queries: membership and equivalence queries. Additionally, we will cover four active learning algorithms: L^*, NL^*, Kearns-Vazirani, Rivest-Schapire.

We evaluated the performance of our tool, dodo-cpp, on a set of common examples for parameterized verification, and compared the results of these algorithms. The main findings of this thesis show that: (1.) . . . , (2.) . . . , (3.) . . . (NEED MORE INFO FROM EXPERIMENT)

Acknowledgments

I could not have undertaken this journey without the support of Christoph Welzel-Mohr. I want to thank Christoph Welzel-Mohr for guiding me during the thesis.

I'd like to acknowledge to Chair of Theoretical Computer Science for providing the foundational knowledge necessary for me to finish this thesis.

Last but not least, thanks also go to my family, which was both supportive and patient.

Danke!
Thank you!
Cam on!

Contents

1. Introduction

As software systems grow in size and permeate more and more areas of our lives. Individuals and organizations use the majority of software in their systems. Thus the reliability and stability of the software testing are of major importance. Simulation and testing can detect bugs but not prove their absence. In such reactive systems, when no function is being computed, termination is usually undesirable. For this reason, we are interested in *property checking* or *model checking*. It has been widely used in various real-world applications, ranging from adaptive model checking to solving real-world problems ([BFL04], [Abd+04]).

We here consider the verification of safety properties similar to the original Regular Model Checking framework, where a program is represented using symbols and finite automata. To be more specific, our goal is to confirm that a given program cannot execute in a way that starts from a set of initial configurations and leads to a set of dedicated bad configurations. In other words, these bad configurations should be not reached during the program's execution. However, this is an undecidable question in general and tools for Regular Model Checking still need to be completed. A solution to this problem was proposed in [Wel23], which utilizes *inductive statements* to ensure that no undesired configuration can be reached from any initial configuration. This means that for every pair of initial and undesired configurations, there is at least one inductive statement that is satisfied by the initial configuration but not by the undesired one. By doing this, it can be concluded that no undesired configuration can be reached.

Over the past decade, there has been a significant increase in the study of automata learning. This field has produced numerous successful applications, such as pattern and natural language recognition, computational biology, data mining, robotics, automatic verification, and even the analysis of music. One can use autoamta learning to acquire a set of inductive statements that are powerful enough to establish a given safety property. The language of these inductive statements serves as proof of the property's correctness. The purpose of this thesis is to collect and analyze empirical data on the performance of learning algorithms such as L*, NL*, Kearns-Vazirani, Rivest-Schapire.

Structure of the thesis

In this thesis, we begin with Chapter 2 by fixing notations and definitions used throughout the thesis. In Chapter 3, we will thoroughly explain the *Regular transition system* and *Inductive statements* as an approach for checking the safety properties of *model checking*. Subsequently, Chapter 4 gives a general introduction to active learning algorithm and their oracles. Furthermore, we will introduce some active learning algorithms that used for our experiments. Chapter 5 will investigate the C++-implemented programm to learn a set of inductive statements from systems called *dodo*. The programm uses not only the *Angluin's algorithm* L^*, but also the NL^*, *Kearns-Vaziran* and *Rivest-Schapi*. After learning process, it visualizes the graphs that can evaluate the *efficiency* and *effectiveness* of these algorithms. Finally, we will summarize and assess our experment results in Chapter 6 and conclude the thesis with Chapter 7.

2. Preliminaries

In this chapter, we introduce some basic notions and definitioms that we use throughout this thesis.

Words and Languages

An *alphabet* Σ is a finite set of symbols. A *word* $u = a_1 \dots a_n$ is a finite sequence of symbols $a_i \in \Sigma$ for $i \in \{1, \dots, n\}$. Σ^* denotes the set of all words over an alphaber Σ. *Regular languages* are those which can be identified by a finite state automaton [Wik23b].

Finite automata

We classify automata into two categories: deterministic and non-deterministic, in order to identify regular languages consisting of finite words.

Definition 2.1: Deterministic finite automaton (DFA)

A DFA is a quintuple $\mathcal{M} = (Q, q_0, \Sigma, \delta, F)$ where Q is a finite set of states with a initial state $q_0 \in Q$. A set of input symbols called the alphabet Σ. A transition $\delta : Q \times \Sigma \to Q$ and a set of final states F. Let $w = a_1 a_2 ... a_n$ be a string over the alphabet Σ. The automaton $\mathcal{M}$ accepts w if a sequence of states, $r_0, r_1, ...r_n$ exist in Q:

- $r_0 = q_0$
- $r_{i+1} = \delta(r_i, a_{i+1}), \textit{for } i = 0, ..., n-1$
- $r_n \in F$

Definition 2.2: Nondeterministic finite automaton (NFA)

A NFA is a quintuple $\mathcal{N} = (Q, q_0, \Sigma, \Delta, F)$ *where* Q*,* Σ *and* F *are as for a DFA. Let*

> *$w = a_1 a_2 ... a_n$ be a string over the alphabet Σ. The automaton $\mathcal{N}$ accepts w if a sequence of states, $r_0, r_1, ...r_n$ exist in Q:*
>
> - $r_0 = q_0$
> - $r_{i+1} \in \Delta(r_i, a_{i+1}), for\ i = 0, ..., n-1$
> - $r_n \in F$

Token passing algorithm

We will provide a simple example to demonstrate how systems are modelled in *regular transition system*. The *token passing* system comprises a linear array of agents where the agent holds a token, and in each step, the current agent can pass the token to its right neighbour. We choose to represent the agent that holds the token as the letter t and the agents that do not hold the token as the letter n.

3. Inductive statements for regular transition system

In the *Regular Model Checking* framework, program configurations are represented as finite words over a pre-determined alphabet Σ. The system comprises a series of starting configurations and the transitions are modelled as the relations mapping configuations to configuations. In Section 3.1, we introduce *Regular transition system* (RTS) —an important framework for infinite state model-checking —to represent the behavior of a system. Section 3.2 will explain how to encode a inductive statement. In addition, we will be presenting the three interpretations that we have utilized in this thesis.

3.1. Regular transition system

Essentially, a *regular transition system* represents a parameterized system $\mathcal{S}$. For example, a *token-passing* system $\mathcal{S}$ with n is the number of agents. We call Σ is the set of alphabets of the system, which indicates the finite states of the agent. For $\mathcal{S}$, $\Sigma = \{n, t\}$. The system begins with initial configurations, defined by a regular language. A sequence of alphabets represents the corresponding state of the agents. In other words, one can understand that the first letter indicates the state $u_1 \in \Sigma$ of the first agent, the second letter indicates the state $u_2 \in \Sigma$ of the second agent, and so on. The states of each agent can be changed by following the system's rules, called the relations. Formally, we call that a *transducer* and define these relations in the form of an NFA as follows:

Definition 3.1: Transducer

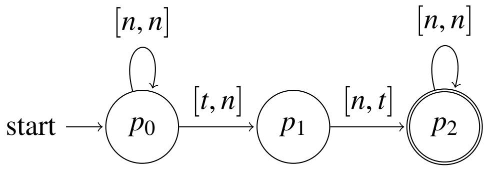
A Σ-Γ*-transducer* $\mathcal{T}$ *is an NFA* $\langle Q, Q_0, \Sigma \times \Gamma, \Delta, F\rangle$*, we denote a relation*

$$[[\mathcal{T}]] = \{\langle u_1 \ldots u_n, v_1 \ldots v_n\rangle \in \bigcup_{n \geq 0} \Sigma^n \times \Gamma^n \mid \langle u_1, v_1\rangle \ldots \langle u_n, v_n\rangle \in \mathcal{L}(\mathcal{T})\}$$

Note that this relationship is only applicable to words that have the same length. Extend *this notation, we define*

$$For\ v \in \Sigma^* : target_{\mathcal{T}}(v) = \{u \in \Gamma^* \mid \langle v, u\rangle \in [[\mathcal{T}]]\}$$

Automaton 3.1: Transducer Γ for Token passing



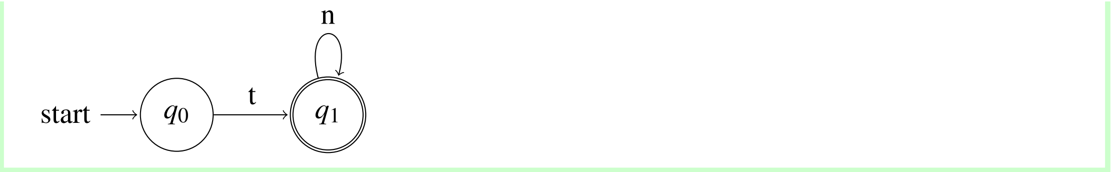
Automaton 3.1 indicates that the token will be transferred from the left agent to the right agent. Once the token reaches the end of the agents, no further transitions can be made to the configuration. We capture the transitions of the *token passing* system via the language $[n, n]^*[t, n][n, t][n, n]^*$. Consider the following scenario: "n n t n" represents a system with four agents, where the third agent currently holds the token. In the next step, only the third agent can transfer the token to the agent on the right. This can be represented as "n n n t". The system terminates since the last agent holds the token.

Definition 3.2: Regular transition system (RTS)

An RTS is a triple $\mathcal{R} = \langle\Sigma, \mathcal{I}, \mathcal{T}\rangle$ where Σ is finite alphabet and $\mathcal{I}$ is an NFA, which represents initial configurations. $\mathcal{T}$ is a Σ-Σ-transducer of the system.

We denote with $\rightsquigarrow_{\mathcal{T}}$ the relation $[[\mathcal{T}]]$ and call a pair $\langle u, v\rangle \in \rightsquigarrow_{\mathcal{T}}$ a transition of $\mathcal{R}$. Moreover, let $\rightsquigarrow_{\mathcal{T}}^*$ denote the reflexive transitive closure of $\rightsquigarrow_{\mathcal{T}}$. We consider $w \in \Sigma^*$ *reachable* on $\mathcal{R}$ if there exist $u \in \mathcal{L}(\mathcal{I})$ with $u \rightsquigarrow_{\mathcal{T}}^* v$. Let $reach(\mathcal{R}) \subseteq \Sigma^*$ denotes all reachable configurations.

Automaton 3.2: NFA $\mathcal{I}$ for Token passing



$\mathcal{L}(\mathcal{I})$ defines the set of initial configurations for the *token passing* as tn*. In other words, the first agent always holds the token, while the following agents do not.

3.2. Inductive statements

In general, we need to determine if a given RTS can produce any undesirable word, which is pre-defined in a reglar set. We call this a *Reachability problem.*

Reachability problem

Besides the *RTS* $\mathcal{R}$, an automaton $\mathcal{B}$ for the regular language that denotes the undesired configurations has been provided. The question at hand is whether it is possible to achieve any undesired configuration in this particular transition system. Formally, we have to compute if *reach*($\mathcal{R}$) $\bigcap \mathcal{L}(\mathcal{B}) = \emptyset$? Because the reachability problem is undecidable, a new approach is needed to ensure no undesired configurations are reached. We are exploring whether there exists a pair of configurations, v and u, where u satisfies all the inductive statements that v satisfies. Indeed, if u is reachable from v, then it will satisfy all the same inductive statements that v did since they are inductive. This method can be used to determine whether an undesired configuration can be reached from an initial configuration. If there is (at least) an inductive statement that v satisfies, but u does not, then it is not possible to reach u from v.

Encoded statements

We shall now proceed to examine the process of how the statements are encoded. We consider the statement pattern "in all configuations of a certain length m either agent i_1 is in state σ_1 or agent i_2 is in state σ_2 or . . . or agent i_k is in state σ_k". In general, the nesccessary information of any statement can be encoded as a function $f : \{1, \ldots, m\} \rightarrow 2^{\Sigma}$, while the set of letters $f(i) \subseteq \Sigma$ corresponds to the states the i-th agent. In other words, each agent can be required by any states or not. Consider the following statement: "In all configurations of length 3, the first agent is in state p or the first agent is in state q." Using a certain method, this statement can be encoded as a function $\{1 \mapsto \{p,q\}, 2 \mapsto \emptyset, 3 \mapsto \emptyset\}$. By simplifying this function to $\{p,q\}\ \emptyset\ \emptyset$, we can express the statement in words.

Without context, these words have no meaning or information. Therefore, we use *interpretation* to understand the statements that we encode. For example, how can we answer the question: "Does the words p q q satisfied the statement $\{p,q\}\ \emptyset\ \emptyset$?". The results can vary depending on the interpretation that we choose to use. With the the interpretation "trap", one verify if $p \in \{p,q\}$, or $q \in \emptyset$, or $q \in \emptyset$. It is necessary that at least one word in the configuration has a value in the inductive statement. While the interpretation "siphon" check if $p \notin \{p,q\}$, and $q \notin \emptyset$, and $q \notin \emptyset$. In other words, it ensures that no configuration word has the same value as the statement.

Definition 3.3: Interpretation

For any RTS $\mathcal{R} = \langle \Sigma, \mathcal{I}, \mathcal{T} \rangle$, we call a pair $\langle \Gamma, \mathcal{V} \rangle$ an Γ-interpretation where Γ is a finite alphabet and $\mathcal{V}$ is a deterministic Σ-Σ-transducer. In the following, we denote $u \models I$ to indicate $\langle u, I \rangle \in [[\mathcal{V}]]$.

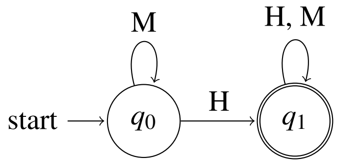
Concrete interpretations

In this thesis, we will explore three interpretations: *trap*, *siphon*, *flow*, and will delve deeper into each of them for a better understanding.

Traps Let fix the size of the instance as n. We define any configuration $u_1 \dots u_n$ the set $\big(u\big) = \bigcup_{1 \leqslant i \leqslant n} \{\langle i, u_i \rangle\}$. For any statement $I_1 \dots I_n$ we define a set $\big(I\big) = \bigcup_{1 \leqslant i \leqslant n} \{i\} \times I_i$. The interpretation of a trap involves connecting a configuration u with a statement I if and only if $\big(u\big) \cap \big(I\big) \neq \emptyset$. Once a configuration has a value in the inductive statement, it can't remove all its values again - it gets "trapped". Formally, $u \models_{\mathcal{V}_{Trap}} I$ if and only if $\big(u\big) \bigcap \big(I\big) \neq \emptyset$. For example, given the configuration $u =$ "n n n t", $v =$ "t n n n" and the statement $I = \{n\}\ \emptyset\ \emptyset\ \emptyset$, one can conclude that $u \models_{\mathcal{V}_{trap}} I$ and $v \not\models_{\mathcal{V}_{trap}} I$.

Automaton 3.3: DFA for trap intepretation

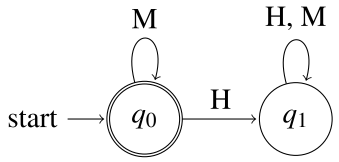
We denote with H all pairs in $\langle \sigma, I \rangle \in \Sigma \times 2^{\Sigma}$ such that $\sigma \in I$ and M all pairs in $\langle \sigma, I \rangle \in \Sigma \times 2^{\Sigma}$ such that $\sigma \notin I$.



Siphon According to the siphon interpretation, none of the values can be part of the configuration that satisfies I, as opposed to the trap interpretation. Formally, $u \models_{\mathcal{V}_{siphon}} I$ if and only if $\big(u\big) \bigcap \big(I\big) = \emptyset$.

Automaton 3.3: DFA for siphon intepretation

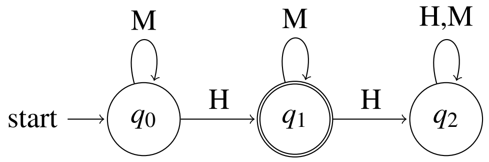
Again, we denote with H all pairs in $\langle \sigma, I \rangle \in \Sigma \times 2^{\Sigma}$ *such that* $\sigma \in I$ *and M all pairs in* $\langle \sigma, I \rangle \in \Sigma \times 2^{\Sigma}$ *such that* $\sigma \notin I$*.*



Flow The third and final interpretation we are interested in is called the flow interpretation $\mathcal{V}_{flow}$. This interpretation requires that the letter of the configuration is included in the set in the same position in the encoded statement. It is important to note that this should occur at only one position. Formally, $u \models_{\mathcal{V}_{siphon}} I$ if and only if $\big|\big(u\big) \bigcap \big(I\big)\big| = 1$. For instance, given the configuration $u =$ "n n n t", $v =$ "t t t t" and the statement $I = \{t\}\ \{t\}\ \{t\}\ \{n, t\}$, one can conclude that $u \models_{\mathcal{V}_{flow}} I$ and $v \not\models_{\mathcal{V}_{flow}} I$.

Automaton 3.3: DFA for flow intepretation

We also denote with H all pairs in $\langle \sigma, I \rangle \in \Sigma \times 2^{\Sigma}$ *such that* $\sigma \in I$ *and M all pairs in* $\langle \sigma, I \rangle \in \Sigma \times 2^{\Sigma}$ *such that* $\sigma \notin I$*.*



Definition 3.4: Inductive statements

For any given Γ-interpretation for $\mathcal{R} = \langle \Sigma, \mathcal{I}, \mathcal{T} \rangle$, we define

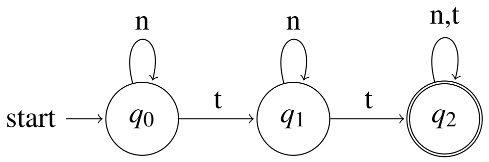
$$Inductive_{\mathcal{V}}(\mathcal{R}) = \{I \in \Gamma^* | \forall u \rightsquigarrow_{\mathcal{T}} \ .if \langle u, I \rangle \in [[\mathcal{V}]] \ then \ \langle v, I \rangle \in [[\mathcal{V}]]\}$$

$$= \{I \in \Gamma^* | \forall u \rightsquigarrow_{\mathcal{T}} \ .if u \models I \ then \ v \models I\}$$

For any RTS $\mathcal{R} = \langle \Sigma, \mathcal{I}, \mathcal{T} \rangle$ and any interpretation $\mathcal{V}$, any inductive statement $I \in Inductive_{\mathcal{V}}(\mathcal{R})$ that is satisfied in one configuration w ($w \models I$}) is also satisfied in all configurations that can be reached from w ($u \models I$ for all $w \rightsquigarrow^*_{\mathcal{T}} u$).

Let's consider the Token passing system example. The token passing algorithm can have a bad property known as "manytoken" (Automaton 3.6). This property refers to all the configurations where more than one agent holds the token simultaneously. In other words, if there are multiple tokens circulating at the same time, it is considered a "manytoken" situation.

Automaton 3.4: DFA $\mathcal{B}$ for "manytoken"



Using the given token-passing system $\mathcal{R}$, a DFA $\mathcal{B}$ and $\mathcal{V}_{trap}$ interpretation, dodo-cpp can learn a regular set of inductive interpretations

$$\{n\} \ \{n\} \ \{n\}^* \ (\{t\} \ \{n\} \ \{n\}^* \mid \{t\}) \subseteq Inductive_{\mathcal{V}_{trap}}(\mathcal{R})$$

For the fixed length of configuration n = 4, all posible inductive statements are

$$\{n\} \ \ \{n\} \ \ \{n\} \ \ \{n\},$$
$$\{n\} \ \ \{n\} \ \ \{n\} \ \ \{t\},$$
$$\{n\} \ \ \{n\} \ \ \{t\} \ \ \{n\}$$

It concludes that $v =$ "$t \ t \ t \ n$" $\in \mathcal{L}(\mathcal{B})$ can not reached from the initial congfiguration $u =$ "$t \ n \ n \ n$" $\in \mathcal{L}(\mathcal{I})$ because u satifies the statement $I = \{n\} \ \ \{n\} \ \ \{n\} \ \ \{t\}$, but v does not.

By this way, we can guarantee that no bad configuations can be reached by checking both origin and the target configuations satisfied all the inductive statements.

Now we will use inductive statements to establish a potential reachability relationship between two configurations.

Definition 3.5: Potential reachability

Let $\mathcal{R} = \langle \Sigma, \mathcal{I}, \mathcal{T} \rangle$ be any RTS and $\langle \Gamma, \mathcal{V} \rangle$ any interpretation. We write $u \Rightarrow_{\mathcal{V}} v$ if and only if $u \models_{\mathcal{V}} v$ for all $I \in target_{\mathcal{V}}(u) \bigcap Inductive_{\mathcal{V}}(\mathcal{R})$.

Lemma 3.1:

Let $\mathcal{R} = (\Sigma, I, T)$ be an RTS, $\langle \Gamma, \mathcal{V} \rangle$ an interpretation, and S a NFA over the alphabet Γ. Then there exists a $\Sigma - \Sigma - transducer$ C such that

$$[[C]] = \left\{ \langle u, v \rangle \in \bigcup_{n \geq 0} \Sigma^n \times \Sigma^n \,|\, \forall I \in \mathcal{L}(S)\,.\,if\ u \models_{\mathcal{V}} I\ then\ v \models_{\mathcal{V}} I \right\} \tag{3.1}$$

Because regular languages are closed under complement, we can define $\overline{C}$ as followings:

Lemma 3.2:

Let $\mathcal{R} = (\Sigma, I, T)$ be an RTS, $\langle \Gamma, \mathcal{V} \rangle$ an interpretation, and S a NFA over the alphabet Γ. Then there exists a $\Sigma - \Sigma - transducer$ $\overline{C}$ such that

$$[[\overline{C}]] = \left\{ \langle u, v \rangle \in \bigcup_{n \geq 0} \Sigma^n \times \Sigma^n \,|\, \exists I \in \mathcal{L}(S)\,.\,if\ u \models_{\mathcal{V}} I\ then\ v \not\models_{\mathcal{V}} I \right\} \tag{3.2}$$

Note that, these Lemmas have been proved in the previous work [Wel23]. We will utilize them later for constructing the equivalent oracle in the implementation Chapter 5.

4. Algorithmic Learning of Finite Automata

Learning automata is a computational model for solving problems, where an agent learns to optimize its behavior by interacting with an unknown environment. The agent, also known as a learner, observes the feedback from the teacher, updates its internal state, and adjusts its actions accordingly. This interaction process between the *Learner* and the *Teacher* is the primary mechanism of learning automata. In the field of automata learning, there are generally two distinct settings: active and passive learning. Passive algorithms are provided with a fixed set of examples consisting of strings that the automaton should either accept or reject. Active algorithms, unlike passive ones, have the ability to expand the set of examples as needed by asking further queries. However, in this thesis, our focus is solely on active learning. We do not introduce passive learning here but refer the interested reader to [CES09].

This chapter intends to offer a comprehensive insight into the learning automata process, specifically discussing the roles and responsibilities of the Teacher and Learner in Section 4.1. In Section 4.2, we will introduce multiple active algorithms that we will use for our experiment.

4.1. The oracles

In this learning scenario, the *Teacher* is proficient in the language being taught and is responsible for answering any questions posed by the learner. The *Learner* is given the opportunity to ask two types of queries - membership and equivalence. Membership queries are used to classify a word based on whether it belongs to the language being taught or not. Equivalence queries, on the other hand, are used to determine whether an assumed automaton is equivalent to the language the *Teacher* has in mind. The learning process continues until the *Teacher* answers an equivalence query positively.

Membership oracle The *Learner* provides a word $w \in \Sigma^*$, the *Teacher* replies "yes" or "no" depending on whether $w \in \mathcal{L}$ or not.

Equivalent oracle The *Learner* conjectures a regular language, typically given as a DFA $\mathcal{M}$, and the *Teacher* checks whether $\mathcal{M}$ is an equivalent description of the target language $\mathcal{L}$

and return "yes", otherwise return an counterexample $u \in \Sigma^*$ with $u \in \mathcal{L}(\mathcal{M}) \iff u \notin \mathcal{L}$ or $u \in \mathcal{L} \iff u \notin \mathcal{L}(\mathcal{M})$.

On equivalent oracle, the *Teacher* can return a positive counterexample or a negative counterexample [Che+17]. A positive counterexample is a missing word in the conjecture but present in the target. The negative one is defined symmetic.

Active learning faces the challenge that the runtime of a learning algorithm is influenced by the quality of counterexamples. If a teacher provides an unnecessarily long counterexample, the learner has no option but to process the entire word to make progress. It is crucial for the *Teacher* to have a clear and specific understanding of the correct hypothesis. Since we know how to implement the *Teacher* to answer the oracles, it is now simple to apply different of learning algorithms.

4.2. Algorithms

A learning algorithm—often called learner—learns a regular target language $\mathcal{L} \subset \Sigma^*$ over an a priori fixed alphabet Σ by actively querying a teacher. We apply several of these algorithms in the course of this thesis.

4.2.1. L*

L* learning automata was introduced by Angluin in 1987 [Ang87], also called Angluin's algorithm. Angluin's algorithm has the ability to learn a regular set which is unknown initially, from any *Teachers*. During the learning process, it stores information in an observation table $O = (S, E, T)$ where $S \subseteq \Sigma^*$ is a nonempty *prefix-closed* [1] set, a finite *suffix-closed* [2] set E, and T : $(S \cup S \cdot A) \cdot E \to \{0, 1\}$ is a mapping that stores the table entries. The algorithm maintains $T(u) = 1$ if and only if u is accepted by the target language for all $u \in (S \cup S \cdot A) \cdot E$.

Let's take a closer look at the inner workings of the *Angluin's algorithm*. For each $row(s)$ of the table, where $s \in S$ denotes a function

$$f_s : E \to \{0, 1\} \ with \ f_s(e) = T(s \cdot e)$$

The overvation table has two properties: *closed* and *consistent*. An observation table is called *closed* provided that for each t in $S \cdot A$ there exists an s in S such that row(t) = row(s). An

[1] A set of strings S is called prefix-closed if: $uv \in S \implies u \in S$

[2] A set of strings S is called suffix-closed if: $uv \in S \implies v \in S$

observation table is called *consistent* provided that whenever s_1 and s_2 are elements of S such that row(s_1) = row(s_2) for all a in A, row($s_1 \cdot a$) = row($s_2 \cdot a$). Once the table is *closed* and *consistent*, we can build a deterministic finite-state acceptor, which also is called *conjecture*, by using the observation table. More precisely, *Angluin's algorithm* constructs the DFA $\mathcal{H} = (Q, q_0, \Sigma, \delta, F)$ where:

$$Q = \{row(s) : s \in S\},$$

$$q_0 = row(\lambda),$$

$$F = \{row(s) : s \in S \;\; and \;\; T(s) = 1\},$$

$$\delta(row(s), a) = row(s \cdot a).$$

Basically these two conditions *closed* and *consistent* guarantee that the transitions is well-defined. The observation table is *closed* ensures that every row in the lower part also occurs in the uper part. In other words, the row labeled by elements of S are the candidates of states of the automaton. *Consistent* condition implies that both words lead to the same state in the automaton, as they cannot be distinguished by any $a \in \Sigma^*$.

The peseudocode 1 presents Algluin's algorithm in pseudocode. Essentialy, in the begin of learning process, the algorithm guarantees that the table are *closed* and *consistent* by repeatly modifiding the columns and also the rows of the table. After every extension of the table, the algorithm fill the table by asking the membership queries for all table entries $u \in (R \cup R \cdot \Sigma) \cdot S$ for which no membership is yet present by asking the *Teacher* the membership queries. If the *Teacher* replies "yes", then set $T(u) = 1$, otherwise $T(u) = 0$. Once this is the case, the observation table satifies the conditions, Angluin's algorithm constructs a conjecture, which it submits to an equivalence query. The learning terminates once the teacher replies “yes” on an equivalence query. However, if the Teacher returns a new counterexample $t \in \Sigma^*$ the algorithm modifies the table by adding t and its prefixes to S and repeats the process by going to line 1.

4.2.2. NL*

In general, a nondeterministic finite automata *NFA* is often preferable to a deterministic finite automata *DFA* due to potentially exponential differences in their sizes [Ozo+05]. Therefore, learning algorithms for nondeterministic finite automata are required. In this section, we will introduce another active learning algorithm called the *NL** algorithm [Bol+09], based on *L**. The *NL** concludes a residual finite-state automata (RFSA), a subclass of nondeterministic finite automata was introduced in the seminar work [DLT01].

Algorithm 1 Algluin's learning algorithm [Ang87]

```
Input: A teacher for a regular language $L \subseteq \Sigma^*$
Initialize the observation table (S, E, T)
Ask membership queries for $\lambda$ and each $a \in \Sigma$
Repeat:
while (S,E,T) is not closed or not consistent do
    if (S,E,T) is not consistent then
        find $s_1$ and $s_2$ in S, and $e \in E$ such that
        $row(s_1) = row(s_2)$ and $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$,
        add $a \cdot e$ to E,
        conducts membership queries.
    end if
    if (S,E,T) is not closed then
        find $s_1$ and $a \in \Sigma$ such that
        $row(s_1 \cdot a)$ is different from $row(s)$ for all $s \in S$,
        add $s_1 \cdot a$ to S,
        conducts membership queries.
    end if
end while
Once (S, E, T) is closed and consistent, make $M = M(S, E, T)$
if the Teacher replies with a counter-example t, then then
    add t and all its prefixes to S
    conducts membership queries.
end if
Util the Teacher replies "yes"
Terminate and return a conjecture $\mathcal{M}$
```

Technically, it is possible to learn an RFSA instead of a DFA by modifying Angluin's algorithm L^* observation table. The proposed method involves selecting *prime rows*[3] as representations of the automaton's states, rather than utilizing *all rows* of the table. The proceed of the NL^* learning algorithm is mainly the same with $L*$. Similar to L^*, it is also repeatedly checked the *RFSA-closed*[3] and *RFSA-consitency*[3] properties, once the both properties are fullfill, it can contruct the conjecture and ask the equivalent query to the teacher.

4.2.3. Kearns-Vazirani

Another active learning algortihm is introduced in this thesis is *Kearns and Vazirani's* algorithm [KV94]. Unlike *Angluin's algorithm* it organizes its data in an ordered binary tree. It aims to minimize the number of membership queries by storing only one representative for each L-equivalence class in the tree. The data are stored in two non-empty set $R, S \subseteq \Sigma$, where R consists of *representatives* that are used to represent the equivalence classes of L. The set S includes *separating words* that are used to verify that two different representatives indeed represent different equivalent classes. More formally, *Kearns-Vazirani's* algortihm keeps a separating word $v \in S$ for any two representatives $u \neq u' \in R$ such that $uv \in L \Leftrightarrow uv \notin L$ is satisfied.

The organization of the binary tree is simple, while the inner nodes are labeled with the word of S, the leaf nodes are labled with words of R. The algorithm labels the root node's with $\epsilon \in S$. The main property is that for each subtree, it places on the subtree's root $v \in S$ and all the $u \in R$ depending on whether $uv \in L$ or not. When $uv \notin L$ u is put in the left subtree. Otherwise, $uv \in L$ u will be put in the right subtree. This procedure is recursively repeated at each subtree until all representatives are put in their own leaf node.

The conjecture of *Kearns-Vazirani's* algorithm is defined following: DFA $\mathcal{H} = (Q, \Sigma, q_o, \delta, F)$. Where the set of states $Q = R$. The final states F consist of all representatives $u \in R$ that are located in the right subtree of the root node. Since ϵ is always an element of R, the initial state $q_0 = \epsilon$.

4.2.4. Rivest-Schapire

The last algorithm we will introduce in this thesis is *Rivest and Schapire* [RS89]. The different of this algortihm to *Angluin's algorithm* is, that uses a *reduced* version of Angluin's observation table that stores exactly one representative per L-equivalence class. The advantages are storing less data and asking less memberships queries. (In fact, this method has originally been introduced by Schapire).

[3] *prime row, RFSA-closed, RFSA-consitency* are defined in [Bol+09]

4.3. Libalf: the Automata Learning Framework

Libalf [Bol+10] is an open-source program library that facilitates the learning of finite automata. It has been extensively used in the experiments conducted in this thesis. Many of the algorithms that were used or developed in later chapters have been integrated into the library. It supports both for active and passive algorithms but in this thesis we only consider the active algorithms.

Internlly, *libalf* represents words as list symbols where each symbol is an integer data type. Thus, the maximal size of an alphabet is 2^{32} or 2^{64} depending on the architecture of the target machine. The libalf library requires the libAmore++ basis library to support the NFA and DFA automaton representation.

5. Implementation

We use automata learning algorithms to solve regular model checking problems and generate inductive statements for the parameterized systems.

5.1. Membership oracle

On a membership oracle, the learner provides a statement and asks the teacher if this statement whether inductive or not. As we described in Definition 3.4, a statement I is *inductive* if, for any transition $v \rightsquigarrow u$ where u satisfies I, u also satisfies the statement. Oone can implement the Membership Oracle by checking the acceptance of $\mathcal{M}$, where $\mathcal{M}$ is an automaton for $\overline{Inductive_{\mathcal{V}}(\mathcal{R})}$ and negating the answer (Algorithm 2). The $\overline{Inductive_{\mathcal{V}}(\mathcal{R})}$ is defined by:

$$\overline{Inductive_{\mathcal{V}}(\mathcal{R})} = \{I \in \Gamma^* \mid \exists u \rightsquigarrow_{\mathcal{T}} w \,.\, u \models I \, and \, w \not\models I\} \tag{5.1}$$

Let $\mathcal{T} = \langle P, \Sigma \times \Sigma, \Delta, p_0, E\rangle$ is a transducer and $\mathcal{V} = \langle Q, \Sigma \times \Gamma, \delta, q_0, F\rangle$ is an interpretation. The automaton of $\overline{Inductive_{\mathcal{V}}(\mathcal{R})}$ is defined by $\langle Q \times P \times Q, \Gamma, \vartriangle, \langle q_o, p_0, q_o\rangle, E \times F \times (Q \setminus F)\rangle$ where

$$\vartriangle(\langle q_1, p, q_2\rangle, I) = \exists\langle\sigma_1, \sigma_2\rangle \in \Sigma \times \Sigma. \;\; (\delta(q_1, \langle\sigma_1, I\rangle),\; \Delta(p, \langle\sigma_1, \sigma_2\rangle),\; \delta(q_2, \langle\sigma_2, I\rangle))$$

The states that are accepted by this automaton when each its parts are satified:

$$\begin{aligned} \delta(q_1, \langle\sigma_1, I\rangle) &\in F \\ \Delta(p, \langle\sigma_1, \sigma_2\rangle) &\in E \\ \delta(q_2, \langle\sigma_2, I\rangle) &\notin F \end{aligned}$$

For every pair of initial word and its reached word through the transducer. Where the initial word is satified by the statement I, the reached word is not. From 5.1 it can guarantee that all statements, that are acepted by this automaton, are non-inductive.

Algorithm 2 Membership oracle

Input: *Statement $\mathcal{I}$*

Output: *True* or *False*

```
begin
  $\mathcal{M} \leftarrow getAutomaton(\overline{Inductive_{\mathcal{V}}(\mathcal{R})})$
  if $\mathcal{I} \in \mathcal{L}(\mathcal{M})$ then
      return $false$;
  else
      return $true$;
  end if
end
```

5.2. Equivalent oracle

When the learner provides a conjecture, the teacher checks if it satisfies the safety property. It it does, the teacher return *true*. Otherwise, the learner receives a counter example and repeates the proocess.

Firstly, we ensure the automaton only accepts inductive statements. We intersect the automaton of $\overline{Inductive_{\mathcal{V}}(\mathcal{R})}$ with the hypothesis. If there exists any non-inductive statement in the hypothesis, we return it as counterexample.

Since hypothesis $\mathcal{H}$ does not accept any non-inductive statement, we will check with the safety problems to make sure that the hypothesis strong enough. Intuitively, the automaton $\mathcal{D}$ in Algorithm 2 contains all pairs from initial and bad words, which is induced by the inductive statements $\mathcal{L}(\mathcal{H})$. In other words, the safety property is that the inductive statements should not induce the initial and bad word. We return true and terminates the algorithm if $\mathcal{L}(\mathcal{D}) = \emptyset$. Otherwise we obtain a counterexample $\langle u_1 \dots u_n,\ v_1 \dots v_n \rangle \in \mathcal{L}(\mathcal{D})$. Intuitively, we can see that $\mathcal{D}$ is the intersection of the automaton $[[C]]$ (Lemma 3.1) and $\mathcal{I} \circ \mathcal{B}$. Since computing $[[\overline{C}]]$ (Lemma 3.2) is effectively, we will construct the automaton for $[[\overline{C}]]$ and complement it. Let $S = \langle P, \Gamma, \Delta, p_0, E \rangle$ is a transducer and $\mathcal{V} = \langle Q, \Sigma \times \Gamma, \delta, q_0, F \rangle$ is an interpretation. The automaton of $[[\overline{C}]]$ is defined by $\langle Q \times P \times Q, \Sigma \times \Sigma, \triangle, \langle q_o, p_0, q_o \rangle, E \times F \times (Q \setminus F) \rangle$ where

$$\triangle(\langle q_1, p, q_2 \rangle, \langle \sigma_1, \sigma_2 \rangle) = \exists I \in \Gamma.\ \ (\delta(q_1, \langle \sigma_1, I \rangle),\ \Delta(p, I),\ \delta(q_2, \langle \sigma_2, I \rangle))$$

Algorithm 3 Equivalent oracle

```
Input: Statement $\mathcal{I}$
Output: $True$, X, or $I \in \Gamma^*$
begin
  $\mathcal{M} \leftarrow getAutomaton(\overline{Inductive_{\mathcal{V}}(\mathcal{R})})$
  if $\mathcal{L}(\mathcal{H}) \cap \mathcal{L}(\mathcal{M}) \neq \emptyset$ then    ▷ Make sure that all statements are inductive
      return $I \in \mathcal{L}(\mathcal{H}) \cap \mathcal{L}(\mathcal{M})$
  end if
  $\mathcal{D} \leftarrow getAutomatonFor(\mathcal{L}(\mathcal{I}) \circ \overset{\mathcal{L}(\mathcal{H})}{\Rightarrow} \circ \mathcal{L}(\mathcal{B}))$    ▷ Check safety property
  if $\mathcal{D} = \emptyset$ then
      return True
  end if
  $\begin{bmatrix} u_1 \\ v_1 \end{bmatrix} \dots \begin{bmatrix} u_n \\ v_n \end{bmatrix} \leftarrow getWordFrom(\mathcal{L}(\mathcal{D}))$
  $I = disprove(\begin{bmatrix} u_1 \\ v_1 \end{bmatrix} \dots \begin{bmatrix} u_n \\ v_n \end{bmatrix})$
  if $I = null$ then
      return X    ▷ throw exception when can not disprove
  end if
  return I
end
```

The states that are accepted by this automaton when each its parts are satified:

$$\begin{aligned} \delta(q_1, \langle\sigma_1, I\rangle) &\in F \\ \Delta(p, I) &\in E \\ \delta(q_2, \langle\sigma_2, I\rangle) &\notin F \end{aligned}$$

5.3. The word problem

We are now trying to locate a counterexample $I \in \mathit{Inductive}_{\mathcal{V}}(\mathcal{R})$ that disproves the given hypothesis. This is done to ensure that $u_1 \dots u_n \models_v I$ and $v_1 \dots v_n \not\models I$, our inductive statements will no longer induce this pair. We can also call I an active counterexample since I is in the target language but was missing in the candicate language. It gives rise to the question whether $I \in \mathit{Inductive}_{\mathcal{V}}(\mathcal{R})$ exists such that $u_1 \dots u_n \models_v I$ and $v_1 \dots v_n \not\models I$. It was previously proven by [Wel23] that this problem is in NP. Moreover, since SAT problem is NP-hard, it can be reduced to SAT.

Flow interpretation In this section, we will extract separating inductive statements using CNF-SAT. The entire formular is a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals. The idea is, we assign each combination of an alphabet $\sigma \in \Sigma$ and the position a variable. In other words, a pair $\langle\sigma, i\rangle$ is a literal which assigns to a variable. The variables have two value: *true* and *false*. Therefore, σ is a part of I_i if and only if the model value of the literal $\langle\sigma, i\rangle$ is true. Firstly, we introduce the helper function

$$\mathit{ExactlyOne}(V) = \bigvee_{v \in V} v \wedge \bigwedge_{v, v' \in V : v \neq v'} \neg(v \wedge v')$$

Intuitively, it generates a set of clauses that ensure that exactly one of the literals evaluate to *true*. Recall that a statement is not inductive if there exists one transition $\left[\begin{smallmatrix} u_1 \\ v_1 \end{smallmatrix}\right] \dots \left[\begin{smallmatrix} u_n \\ v_n \end{smallmatrix}\right]$ that is accepted by transducer $\mathcal{T}$ for which holds that $x_1 \dots x_n \models I_1 \dots I_n$ and $y_1 \dots y_n \not\models_{\mathcal{V}_{flow}} I_1 \dots I_n$. Formally, we add these clauses to the formular:

$$\mathit{ExactlyOne}\Big(\bigcup_{1 \leq i \leq n} \{\langle u_i, i\rangle\}\Big) \tag{5.2}$$

and

$$\neg\mathit{ExactlyOne}\Big(\bigcup_{1 \leq i \leq n} \{\langle v_i, i\rangle\}\Big) \tag{5.3}$$

Clause 5.2 ensures that there is exactly one $1 \leq i \leq n$ such that $x_i \in I_i$. On the other hand, clause 5.3 guarantees that either there is no or more than one $1 \leq i \leq n$ such that $x_i \in I_i$. Semantically, we define a state $\langle l, q, k\rangle \in \{0,1\} \times Q_{\mathcal{T}} \times \{0,1,2\}$ corresponds to the observation that one can reach the state q of $\mathcal{T}$ with a word $\left[\begin{smallmatrix}u_1\\v_1\end{smallmatrix}\right] \dots \left[\begin{smallmatrix}u_n\\v_n\end{smallmatrix}\right]$ such that there are k many indices i where $x_i \in I_i$, on the other hand, there are l many indices j where $y_j \in I_j$. We need to ensure that each pair $[x, y]$ we consider is accepted by the *transducer* $\mathcal{T}$. Additionally, in the final step should not result in the same configuration for both the source and target. To achieve this, we encode the state product as literals and define the proposition formula as follows:

$$
\begin{aligned}
&\bigvee_{q_0 \in Q_0^{\mathcal{T}}} \langle\langle 0, q_0, 0\rangle, 0\rangle \wedge \neg \bigvee_{f \in F_{\mathcal{T}}} \langle\langle 1, f, 0\rangle, n\rangle \vee \langle\langle 1, f, 2\rangle, n\rangle \\
&\wedge \bigwedge_{1 \le i \le n,\ \left\langle q, \left[\begin{smallmatrix}x\\y\end{smallmatrix}\right], p\right\rangle \in \Delta_{\mathcal{T}}}
\left(\begin{array}{c}
\langle\langle 0, q, 0\rangle, i\rangle \wedge \langle x, i+1\rangle \wedge \langle y, i+1\rangle \implies \langle\langle 1, p, 1\rangle, i+1\rangle \\
\wedge\langle\langle 0, q, 1\rangle, i\rangle \wedge \langle x, i+1\rangle \wedge \langle y, i+1\rangle \implies \langle\langle 1, p, 2\rangle, i+1\rangle \\
\wedge\langle\langle 0, q, 2\rangle, i\rangle \wedge \langle x, i+1\rangle \wedge \langle y, i+1\rangle \implies \langle\langle 1, p, 2\rangle, i+1\rangle \\
\wedge \bigwedge_{k \in \{0,1\}} \left(\begin{array}{c}
\langle\langle k, q, 0\rangle, i\rangle \wedge \neg\langle x, i+1\rangle \wedge \langle y, i+1\rangle \implies \langle\langle k, p, 1\rangle, i+1\rangle \\
\wedge\langle\langle k, q, 1\rangle, i\rangle \wedge \neg\langle x, i+1\rangle \wedge \langle y, i+1\rangle \implies \langle\langle k, p, 2\rangle, i+1\rangle \\
\wedge\langle\langle k, q, 2\rangle, i\rangle \wedge \neg\langle x, i+1\rangle \wedge \langle y, i+1\rangle \implies \langle\langle k, p, 2\rangle, i+1\rangle
\end{array}\right) \\
\wedge \bigwedge_{l \in \{0,1,2\}} \left(\begin{array}{c}
\langle\langle 0, q, l\rangle, i\rangle \wedge \langle x, i+1\rangle \wedge \neg\langle y, i+1\rangle \implies \langle\langle 1, p, l\rangle, i+1\rangle \\
\wedge\langle\langle 0, q, l\rangle, i\rangle \wedge \langle x, i+1\rangle \wedge \neg\langle y, i+1\rangle \implies \langle\langle 1, p, l\rangle, i+1\rangle
\end{array}\right) \\
\wedge \bigwedge_{k \in \{0,1\}\ l \in \{0,1,2\}} \left(\langle\langle k, q, l\rangle, i\rangle \wedge \neg\langle x, i+1\rangle \wedge \neg\langle y, i+1\rangle \implies \langle\langle k, p, l\rangle, i+1\rangle\right)
\end{array}\right)
\end{aligned}
\tag{5.4}
$$

To solve the (CNF-)SAT problem we need to convert the entire of logical formular 5.2, 5.3 and 5.4 in the CNF form, which can be achieved by applying DeMorgan's laws [Wik23a].

A polynomial time algorithm for the word problem for $\mathcal{V}_{trap}$ and $\mathcal{V}_{siphon}$ We focus on $\mathcal{V}_{trap}$ since the arguments for $\mathcal{V}_{siphon}$ are analogous. Pseudocode 4 demonstrates how to find the separating statement for trap interpretation within polynomial time. We begin with the statment $I = \Sigma \backslash \{y_1\} \dots \Sigma \backslash \{y_n\}$. If a transition $\left[\begin{smallmatrix}x_1\\y_1\end{smallmatrix}\right] \dots \left[\begin{smallmatrix}x_n\\y_n\end{smallmatrix}\right]$ exists such that $x_1 \dots x_n$ satisfies the current statement and $y_1 \dots y_n$ does not, then remove x_i from the i-th letter of the statement for all $1 \le i \le n$. To prove that this approach can be computed in polynomial time, refers to [Wel23].

The language we are learning can have a much exponentially larger alphabet than the RTS. However, *Libalf* - the software we are using - allows us to start the learning process with a smaller alphabet, which we can expand later if we need to. So, we begin with an empty alphabet and gradually add letters from 2^{Σ}.

Algorithm 4 Disprove (polymial time algorithm for trap)

```
Input: $\begin{bmatrix}x_1\\y_1\end{bmatrix}\ldots\begin{bmatrix}x_n\\y_n\end{bmatrix}$ and transducer $\mathcal{T}$
Output: Inductive statement I
begin
for $i = 1; i \leq n; i = i + 1$ do
    $I_i = \Sigma \backslash \{y_i\}$
end for
while $\langle v, I \rangle \in \mathcal{L}(\mathcal{V}_{trap})$ do
    if $\exists \begin{bmatrix}x_1\\y_1\end{bmatrix}\ldots\begin{bmatrix}x_n\\y_n\end{bmatrix}$ where $u_1 \ldots u_n \models I$ and $v_1 \ldots v_n \not\models I$ then
        for $i = 1; i \leq n; i = i + 1$ do
            $I_i = I_i \backslash \{u_i\}$
        end for
    else
        return I
    end if
end while
return $\emptyset$
end
```

6. Experiments

6.1. Case studies

Because the main work of this thesis is compare and evaluate performance between of algorithm, we just consider some simple case studies.

Dijkstra's algorithm for mutual exclusion with a token The example illustrates a mutual exclusion algorithm [FO97] for agents forming a ring. They pass around a single token as a semaphore for a critical region.

Other mutual exclusion algorithms Additionally, we also consider the mutual exclusion algorithms of the standard bakery algorithm [Che+17].

Dining philosophers Atomic

Cache coherence protocols When checking for cache coherence protocols, it's important to ensure that there aren't two different versions of the same data point present in the cache. In this regard, we only analyze the protocol MESI. We also examine different custom safety properties for each of these protocols. The models are babse on [Del00].

Leader election HERMAN

Token passing In conclusion, this thesis presents several examples that demonstrate token passing algorithms that we previously introduced in Chapter 2.

6.2. Dodo-cpp

Dodo-cpp is a program that has three interpretations, namely trap, siphon, and flow. It runs for every system with four algorithms, which are L^*, NL^*, Kearns and Vazirani, Rivest-Schapire. After that, it plotted the graphs to compare the learned time of theses algorithms.

After running the algorithms, Dodo-cpp plots graphs to compare the time taken for each algorithm to learn. To make the data easier to analyze, the program uses the mathplot library [Hun07] for graph plotting.

6.3. Graphs

The experiments in this thesis were running on x64 Ubuntu 22.04 system with 12th Gen Intel(R) Core(TM) i7-12700F processor and 16GiB memory.

For all Figures, we use KV as an abbreviation for Kearns and Vazirani's algorithm and RS for Rivest-Schapire's algorithm. All of the algorithms we used to learn produced the same results, regardless of their configuration and whether they could learn. However, the learning time for each algorithm varied. In the case of learning for a token-passing system using Trap and TrapSAT [1] interpretation and the "notoken" property, Figure 6.1 shows that the inductive statements can be learned (marked with blue bars). We can also compare the learning time between the algorithms to determine which one is more effective.

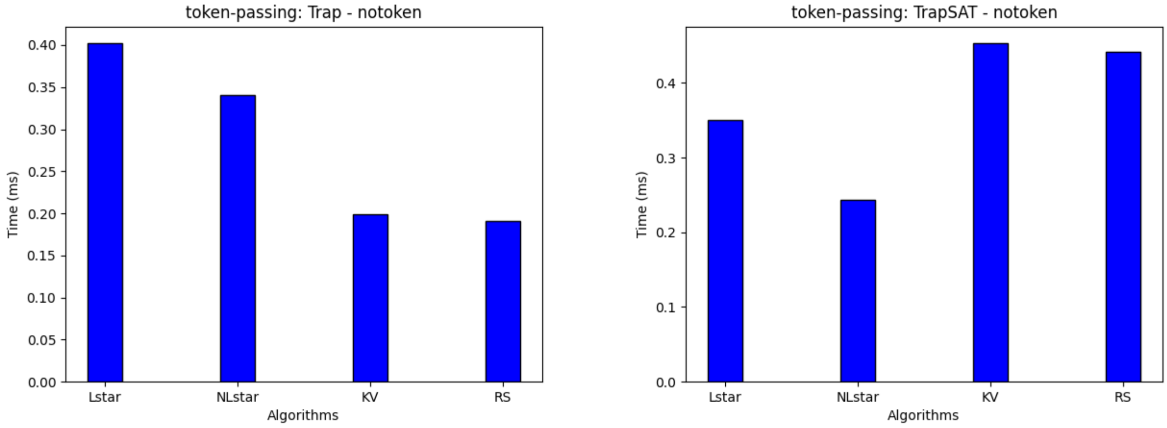


Figure 6.1.: *Comparing the learning time of all algorithms; Result: success; Interpretation: Trap, TrapSAT; System: token-passing; property: notoken.*

In Figure 6.2, we cannot learn using the Siphon and SiphonSAT[1] interpretation. We denote these cases with red bars.

[1] Use SAT-Sovler for the word problem.

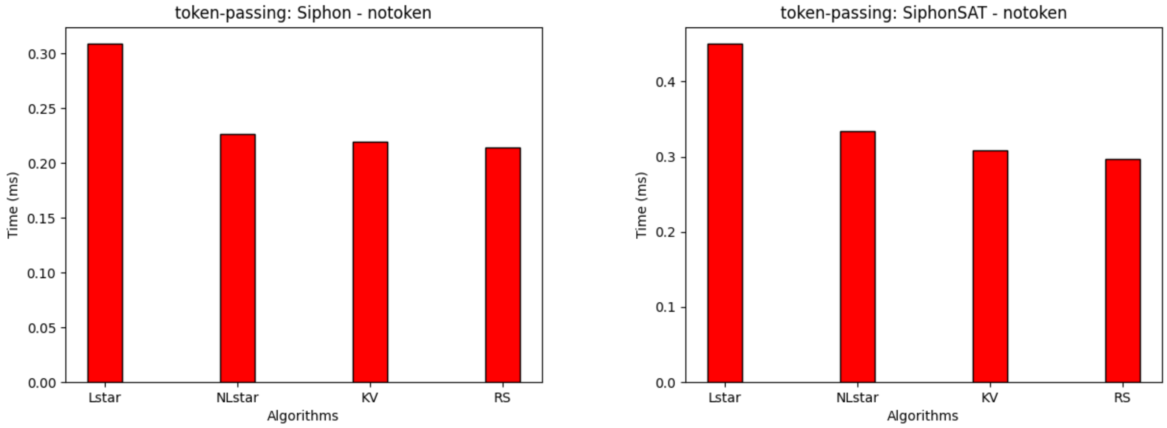


Figure 6.2.: *Comparing the learning time of all algorithms; Result: fail; Interpretation: Siphon, SiphonSAT; System: token-passing; property: notoken.*

6.4. Evaluating

In Figure 6.8, we compare the number of memeber ship queries that are asked by each algorithm. Because in our algorithm, we add the alphabet gradually and it causes membership queries of all algorithm are the same.

we can not say for which algorithm is the best for all cases but at least we know in specifics configuration which one is better.

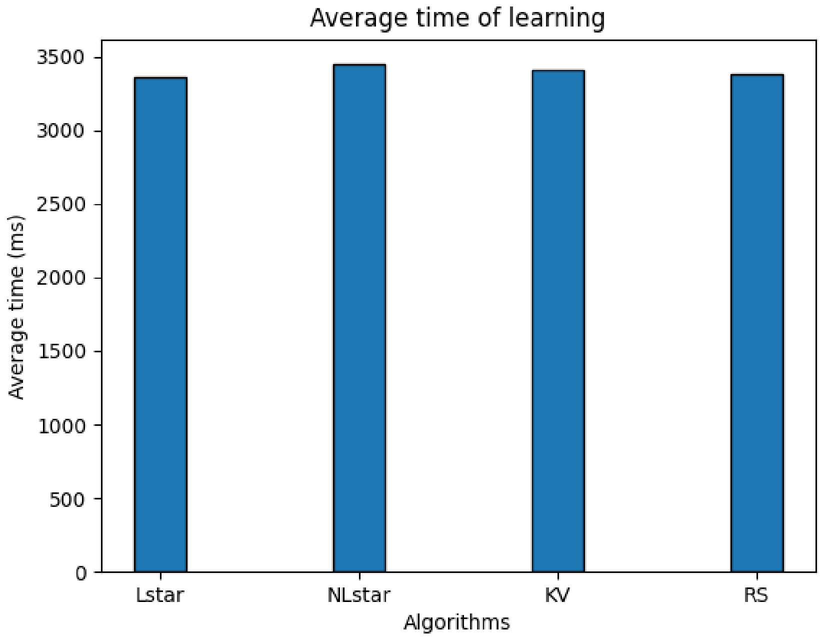


Figure 6.3.: *Comparing the average learning time of all algorithms.*

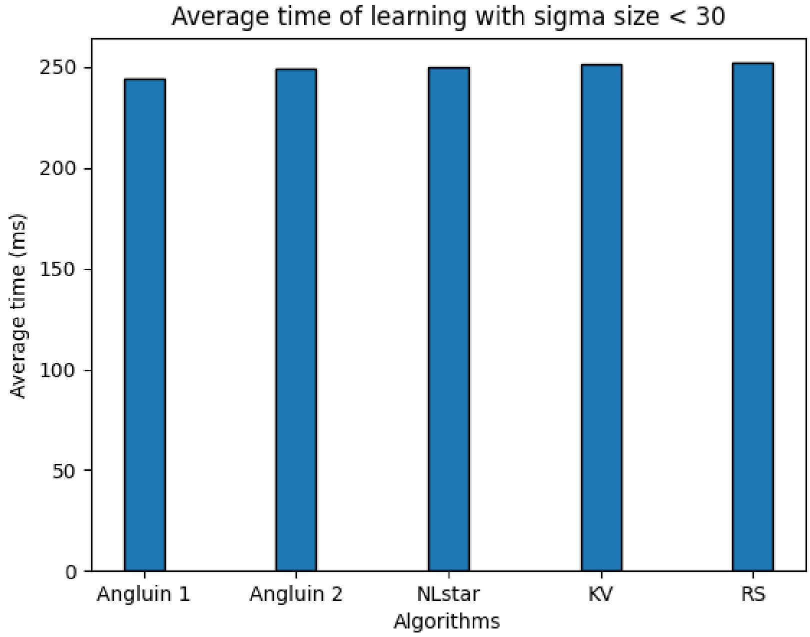


Figure 6.4.: *Comparing the average learning time of all algorithms for all configurations with have |Σ| < 30.*

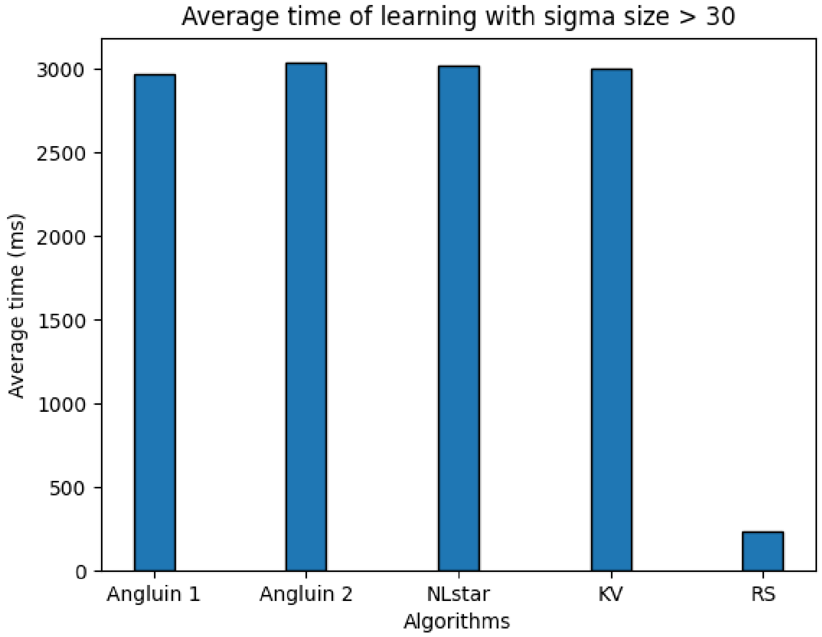


Figure 6.5.: *Comparing the average learning time of all algorithms for all configurations with have |Σ| ≥ 30.*

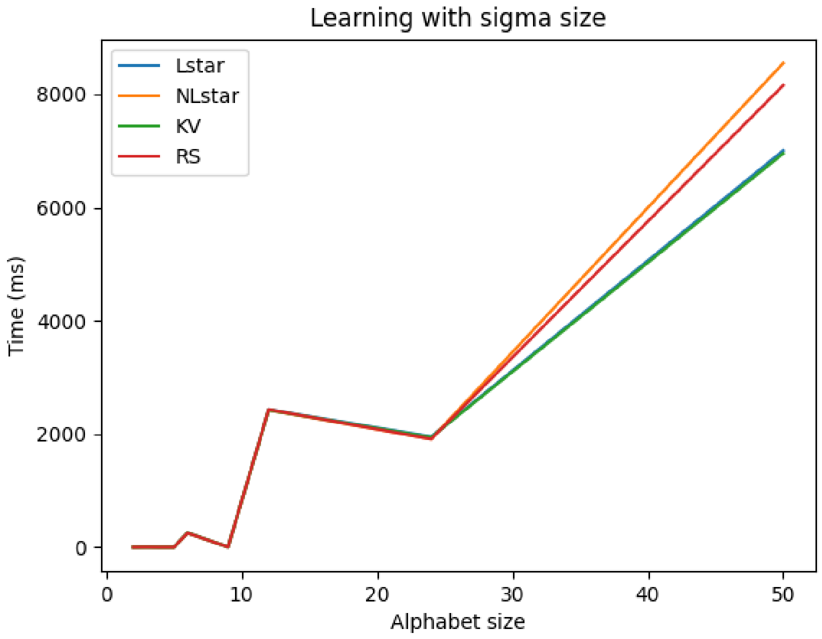


Figure 6.6.: *Comparing the average learning time of all algorithms corresponds to* |Σ|.

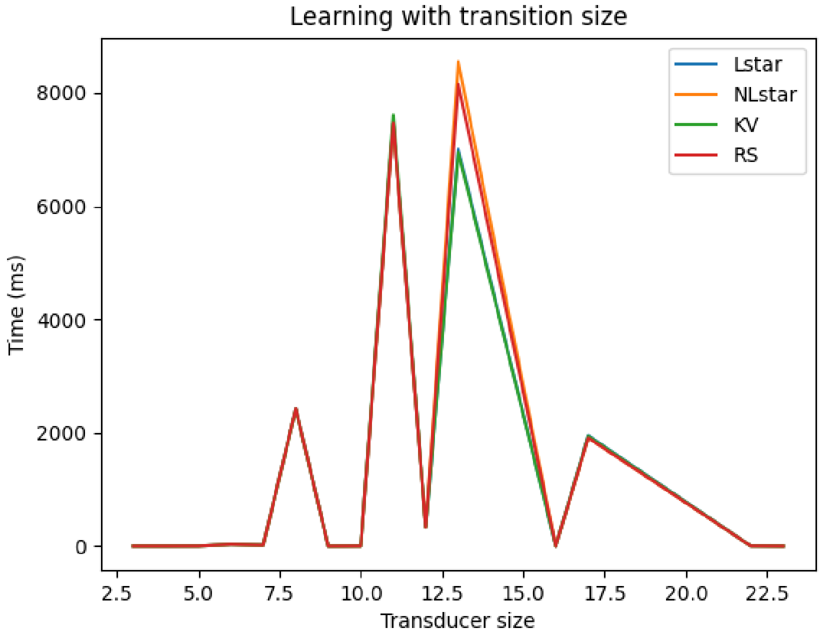


Figure 6.7.: *Comparing the average learning time of all algorithms corresspond to transducer size.*

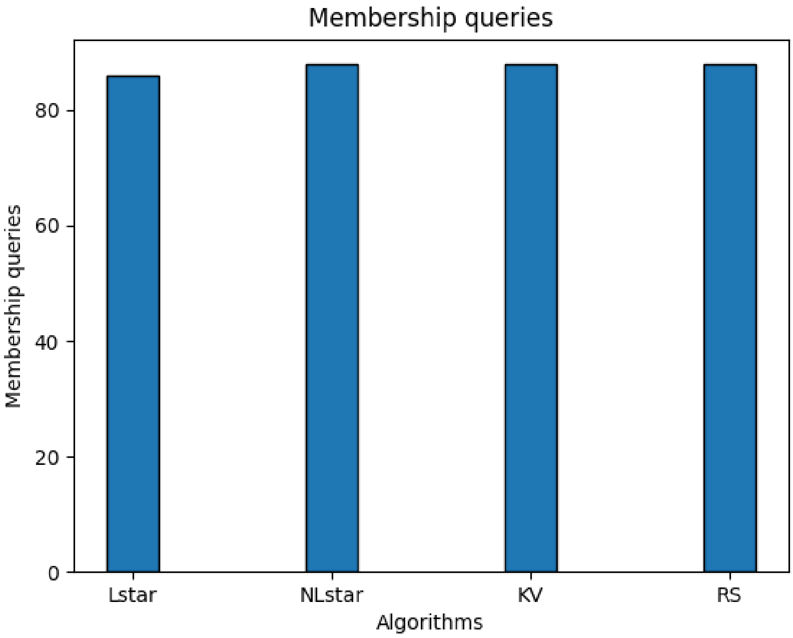


Figure 6.8.: *Comparing the average membership queries of all algorithms.*

7. Conclusion

We studied Regular Model Checking of safety properties. We evaluated the performance of our algorithms based on a prototype implementation.

NEED MORE INFO FROM EXPERIMENT

Open Questions and Future Research

NEED MORE INFO FROM EXPERIMENT

A. Experiments results

A.1. Dijkstra's algorithm for mutual exclusion with a token

A.2. Other mutual exclusion algorithms

Name	Property	Interpretations	Algorithms	Result	Time	Language Size
Berkeley	exclusiveexclusive	Trap	Lstar	x	2.472ms	-
			NLstar	x	2.547ms	-
			KV	x	2.77ms	-
			RS	x	2.254ms	-
		Siphon	Lstar	x	1.829ms	-
			NLstar	x	1.978ms	-
			KV	x	1.67ms	-
			RS	x	1.688ms	-
		Flow	Lstar	x	5.556ms	-
			NLstar	x	5.435ms	-
			KV	x	5.484ms	-
			RS	x	5.428ms	-

Table A.1.

Name	Property	Interpretations	Algorithms	Result	Time	Language Size
Berkeley	exclusiveunowned	Trap	Lstar	v	27.835ms	6
			NLstar	v	25.548ms	6
			KV	v	25.213ms	6
			RS	v	25.961ms	6
		Siphon	Lstar	x	1.19ms	-
			NLstar	x	1.121ms	-
			KV	x	1.091ms	-
			RS	x	1.091ms	-
		Flow	Lstar	x	5.627ms	-
			NLstar	x	5.916ms	-
			KV	x	5.636ms	-
			RS	x	5.861ms	-

Table A.2.

Name	Property	Interpretations	Algorithms	Result	Time	Language Size
Berkeley	exclusivenonexclusive	Trap	Lstar	v	33.441ms	6
			NLstar	v	31.765ms	6
			KV	v	30.473ms	6
			RS	v	30.338ms	6
		Siphon	Lstar	x	1.953ms	-
			NLstar	x	1.74ms	-
			KV	x	1.682ms	-
			RS	x	1.664ms	-
		Flow	Lstar	x	5.616ms	-
			NLstar	x	5.48ms	-
			KV	x	5.422ms	-
			RS	x	5.85ms	-

Table A.3.

Name	Property	Interpretations	Algorithms	Result	Time	Language Size
Berkeley	deadlock	Trap	Lstar	v	0.219ms	1
			NLstar	v	0.094ms	1
			KV	v	0.09ms	1
			RS	v	0.079ms	1
		Siphon	Lstar	v	0.105ms	1
			NLstar	v	0.089ms	1
			KV	v	0.08ms	1
			RS	v	0.078ms	1
		Flow	Lstar	v	0.118ms	1
			NLstar	v	0.083ms	1
			KV	v	0.08ms	1
			RS	v	0.078ms	1

Table A.4.

A.3. Dining philosophers

A.4. Cache coherence protocols

A.5. Leader election

A.6. Token passing

List of Figures

List of Algorithms

List of Tables

Bibliography

[Abd+04] P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. “A Survey of Regular Model Checking.” In: *CONCUR 2004 - Concurrency Theory*. Ed. by P. Gardner and N. Yoshida. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 35–48.

[Ang87] D. Angluin. “Learning regular sets from queries and counterexamples.” In: *Information and Computation* 75.2 (1987), pp. 87–106. ISSN: 0890-5401. DOI: `https://doi.org/10.1016/0890-5401(87)90052-6`.

[BFL04] S. Bardin, A. Finkel, and J. Leroux. “FASTer Acceleration of Counter Automata in Practice.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by K. Jensen and A. Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 576–590.

[Bol+09] B. Bollig, P. Habermehl, C. Kern, and M. Leucker. “Angluin-Style Learning of NFA.” In: *International Joint Conference on Artificial Intelligence*. 2009.

[Bol+10] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon. “libalf: The automata learning framework.” In: *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer. 2010, pp. 360–364.

[Bou+00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. “Regular model checking.” In: *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer. 2000, pp. 403–418.

[CES09] E. M. Clarke, E. A. Emerson, and J. Sifakis. “Model checking: algorithmic verification and debugging.” In: *Communications of the ACM* 52.11 (2009), pp. 74–84.

[Che+17] Y.-F. Chen, C.-D. Hong, A. W. Lin, and P. Ruemmer. *Learning to Prove Safety over Parameterised Concurrent Systems (Full Version)*. 2017. arXiv: `1709.07139 [cs.LO]`.

[Del00] G. Delzanno. “Automatic verification of parameterized cache coherence protocols.” In: *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer. 2000, pp. 53–68.

[DLT01] F. Denis, A. Lemay, and A. Terlutte. “Residual Finite State Automata.” In: *STACS 2001*. Ed. by A. Ferreira and H. Reichel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 144–157. ISBN: 978-3-540-44693-4.

[FO97] L. Fribourg and H. Olsén. “Reachability sets of parameterized rings as regular languages.” In: *Electronic Notes in Theoretical Computer Science* 9 (1997), p. 40.

[Hun07] J. D. Hunter. “Matplotlib: A 2D graphics environment.” In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: `10.1109/MCSE.2007.55`.

[KV94] M. J. Kearns and U. Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[Ozo+05] R. Ozols, R. Freivalds, L. Mancinska, and M. Ozols. “Size of Nondeterministic and Deterministic Automata for Certain Languages.” In: *FCS*. 2005, pp. 169–175.

[RS89] R. L. Rivest and R. E. Schapire. “Inference of finite automata using homing sequences.” In: *Proceedings of the twenty-first annual ACM symposium on Theory of computing*. 1989, pp. 411–420.

[Wel23] C. Welzel-Mohr. “Inductive Statements for Regular Transition Systems.” In: 2023.

[Wik23a] Wikipedia contributors. *De Morgan’s laws — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-December-2023]. 2023.

[Wik23b] Wikipedia contributors. *Regular language — Wikipedia, The Free Encyclopedia*. [Online; accessed 19-December-2023]. 2023.